LISPの演算子について(計算アルゴリズムの基礎理論)

AUTHOR(S):

柳瀬, 龍郎

CITATION:

柳瀬, 龍郎. LISPの演算子について(計算アルゴリズムの基礎理論). 数理解析研究所講究録 1987, 625: 64-70

ISSUE DATE:

1987-05

URL:

http://hdl.handle.net/2433/99965

数理解析研究所講究録
625巻 1987年 64-70

LISPの演算子について

福井大学　柳瀬龍郎

前書き

効率を重視するあまり，副作用を積極的に利用して，手続型プログラムとしての特徴を強く持ちすぎたLISPは「ノイマンネックの解消」と言う課題を課せられた現在，その莫大な遺産を引き継ぐためにはどうすればよいか．この報告では，このことについてのひとつの提案を試みる．

Backus(1) によって提唱されたFPは，副作用を利用しない純粋な関数型システムである．しかし，効率の問題は別にしても，演算子に結び付けられる引数がわかりにくいことや，引数を常に一つのリストとして与える必要があることなどが，現実的な場面で，今一つ使いにくさを感じさせる原因となっている．一方，LISPはその歴史が古く，現在でも様々な場面において利用されている．しかし，データフローやマルチプロセッサなどの新しい計算機アーキテクチャをその動作環境の前提とした場合，すっきりとは馴染まない部分がでてくる．すなわち，PROG，SET ，CONDなどの副作用や逐次処理を陰に陽に前提とした，演算子が含まれていることである．言い換えれば，LISPはこれらの仕様故にノイマン型アーキテクチャを前提にした言語システムになってしまったのである．むろん，それはそれでよいのであろうが，コンパイラと言う高速化のための「必要悪」（少なくとも個人的にはそのように感じる）の存在意義を認めるなら，アーキテクチャをノイマンから変化させたとき「COMMON LISP 」とは違う方向があってもよいのではないだろうか．それなら，というわけでノイマンネックを持たない新しいアーキテクチャを前提にした言語システムを考えればよいことになるが，LISPの遺産があまりにも大きく，また，LISPそのものの基本的な使い易さもあってやはり捨て難い．そこで『LISPに非ノイマン型のアーキテクチャに対する特徴をもたせながらも，当面はLISPプログラムをそのまま走らせることが出来るようにし，将来的にはアーキテクチャに馴染まない仕様は順次削除して行く．』といったような方針を

取ることは出来ないものであろうか．むろん，新しい特徴を持たせる際に，何等かの系統だった方針を持って行うことが必ず必要で，例えば，単に並列評価の為の命令や構文を無秩序に追加して済ませるような事は，避けなければならない．

以上のような視点にたって，”偉大な”LISPの原型はそのままとどめるとしても，FPの形式定義を与えるFFP に似た構文を導入し，新しい関数型としての特徴をもたせることによってノイマンアーキテクチャに捕われないプログラミンが可能な言語システムの可能性を以下において検討する．

本論

一言でいえば，LISPにおいて評価されるＳ表現において，FFP に似た関数構文の使用を許そうというのである．そして，LISPに既に存在する，副作用や逐次処理等を暗黙の前提とする構文を，当面は許すとしても，いずれ版を重ねるにつれて，使用することを抑制もしくは禁止して行く．このことにより，関数型としての特徴を保ちつつ，従来のLISPやFPよりも柔軟なプログラミングが可能なシステムを構築する．以下においてこれをFL（仮称）と呼ぶ．新しく導入されるのは，幾つかの演算子，及び，演算子を組み合わせてその合成式を作るための演算構成子である．表１に示すような演算構成子，表２に示すような演算子を導入し表３に示すような演算子などを削除する．

表１	表２	表３
新演算構成子	新演算子	削除もしくは禁止する演算子
COMPO	DEPRG	SET
CONST	DISTR	PROG
RINSERT	DISTL	（従来型の）COND
APALL	ROTR	DEFUN
FIF	ROTL	（etc ）
WHILE	ID	
BITU	引数選択子	

これらの新しく導入される構文を用いると具体的にはどのようなＳ表現が得られるのであろうか．此のことについて，つぎに考えてみる．

例題

あまり実際的な例ではないが，LISPで評価されるＳ表現が次式であれば

(CAR (CDR (CDR (CDR '(W X Y Z)))))

新しい演算構成子COMPO を使えば，FLでは，

((COMPO CAR CDR CDR CDR) '(W X Y Z))

となり，（COMPO . . . ）の変わりに｛. . . ｝をつかって

({CAR CDR CDR CDR} '(X Y Z))

と表現することも可能である．この”COMPO ”は引数への演算子の適用順序を保存するので副作用がもしあればその順序も保証される．

また，最近のLISPでは並列評価のためにSEPARATE,QLAMBDA,PCALL,etcといった様々な構文が導入されているが，このFLでは表３に示す”CONST ”を用いる．すなわち，

((CONST f_1 f_2 ... f_3) '(X Y Z))

は共通の引数 '(X Y Z)に演算子 f_1，f_2 . . . f_3 を各々独立に適用し，並列に評価して別々に結果を返す．ただし全体としては，結果は１本のリストにして返される．たとえば，

((CONST CAR CDR) '(X Y Z))

は評価されて

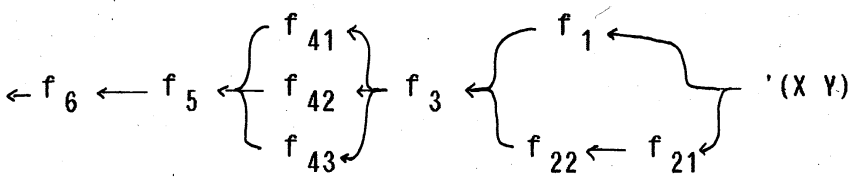
(X (Y Z))

となる．また，この（CONST . . . ）は［. . . ］で省略表現できることにすれば，より簡潔にプログラムを表現できる．

COMPO とCONST を組み合わせて使えば次のようなＳ表現が許されることになる．

({f_6 f_5 [f_{43} f_{42} f_{41}] f_3 [{f_{22} f_{21}} f_1]} '(X Y))

このＳ表現は，次のような関数の関係を持っている．



評価は右から左に向かって逐次行われ，上下にならんでいる関数については，独立して並列に評価される．

表１のRINSERT やその他の演算構成子はFPの場合と同様の意味を持つ．また，表２の引数選択子は関数に直接引数を与えるために必要であり，またDEPRG は，PROGやDEFUN のような局所変数を必要とする定義演算子の代りをなすものである．

プログラム代数について

FPには様々な代数規則があるが，ここでのシステムにも同様な性質がみられる．例えば，FPにおいては

h・（p→f；g）＝p→h・f；h・g

が成り立つが，演算構成子　FIF を使ったFLのＳ表現では，

```
({CAR (FIF {ZEROP 1} 2 3)}  (0 ( a b c) (d e f)))
```

は

```
((FIF {ZEROP 1} {CAR 2} {CAR 3}) (0 (a b c) (d e f)))
```

と同じ結果となる．また

```
({[1 2] cdr} '(X Y Z))
```

は

```
([{1 cdr} {2 cdr}] '(X Y Z))
```

と同じ値(Y Z) を返す．

引数の個数に付いて

FPの場合その引数は一本のリストと決まっている．このため，FLにFPの演算子をそのまま導入すれば従来の演算子と引数の整合が取れない．例えば，接合についてFP型演算子なら

(FAPPEND ((X Y Z) (S T U))) ⟹ (X Y Z S T U)

であり，一つの引数ですむ．またLISPの場合なら

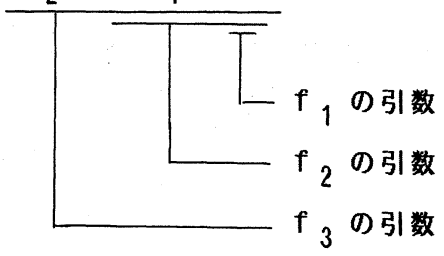
(APPEND '(X Y Z) '(S T U)) ⟹ (X Y Z S T U)

となり，複数の引数が必要となる．現段階のFLではいずれの演算子も使用を許しているが，同様の結果を得るのに，互いに似たような演算子を用意するのは，混乱を招く恐れがあり，いずれかに統一したほうが良いと思われるが，問題のある部分である．

引数の評価について

FPならいずれにしても引数の評価は行われないが，FLにおいて，COMPO により関数の合成を行った場合，引数の評価のメカニズムが問題になってくる．通常のＬＩＳＰなら，演算子の種類によって引数を評価するか否かを，その演算子をふくむＳ表現が評価されるとき決定して評価をすればよい．別の言い方をすれば，引数評価の連鎖によってプログラムの流れが決定されるようになっている．たとえば，引数Ｘに関数 f_1，f_2，f_3 を順に適用するには

$(f_3\ (f_2\ (f_1\ X)))$



のようにする必要があり，しかも，（ｆ１ではない）f_3 と f_2 が共に“引数を評価する”種類の関数（演算子）でなければならない．つまり，LISPプログラミングをしようとすればプログラムとデータが明確に区別されていないために，演算子と引数，すなわちプログラムとデータの『混合物』による『積み木』によって，プログラムの流れと構造を同時に制御することを強いられるのである．このために，数多あるLISPプログラムは，ほとんど，PROGをつかってFORTRAN （風）プログラムミングされたものばかりである．まさに“はじめにPROGありき”である．それに対し，FLではプログ

ラム，すなわち演算子の合成によってのみ，その流れと構造を決定することを理想としている．例えば，次式のように．

（｛f_3 f_2 f_1｝ X） 引数はXのみ．関数はf_1，f_2，f_3の順に実行．

（［f_3 f_2 f_1］ X） 引数はXのみ．関数の実行は並列．

ここで，次のような問題が起こってくる．すなわち，f_1,...,f_3がLISPのいわゆるSUBR，EXPSUBR 関数の場合，あるいは，合成演算式の中に，それらがいくつか含まれている場合，xの評価を(i) 行なうのかどうか，(ii)行うとすれば何時行なうのか，(iii) 合成式に含まれる複数個のSUBR関数に対し，そのつど評価を行うのかどうか，等々．

この問題について，基本的には次のように考える．『システム全体としてはLISPである．しかし，演算子は合成式が許される．』と．そこで，FLでは，具体的には演算子の合成式に引数評価を要求する演算子が現れた時点でただ一度，引数を評価する事にすればよい．例えば，

（{{f_n ... f_3} f_2 f_1} ....）

なら，f_1，f_2，．．．f_n の順に演算子を引数に適用し，評価して行き，その途中で引数評価を要求する演算子が現れれば，その最初の演算子に対してのみ引数を評価する．あとは，演算子が何であろうと引数は評価されない．こうしておけば，少くとも既存のLISPプログラムはそのまま実行可能である．

まとめ

LISPに，合成演算子を許す統語規則を導入することにより，ノイマンネックを前提としない新しいプログラミン言語の可能性を検討した．議論の余地は未だ多く残されているが，今後，基礎的な議論を進めると共に，さまざまな例題，あるいは，応用プログラムを書きながら仕様を固めて行く予定である．

文献

Backus, J. W. "Can Programming be liberated from the vonNewman style? A functional style and its algebla of programs ." CACM ,August 1978.

APPENDIX

例題

(1) ({FID [FCDR CDR]} '(X Y Z)) ⟹ (((W X Y Z)) (X Y Z))

(2) ({CAR CADR} (CDR '(W X Y Z))⟹ (X Y)

```
(3) (DEPRG HANOI { [(FQUOTE a) (FQUOTE b) (FQUOTE c) 1]})
      (DEPRG FHANOI
              (FIF {ZEROP FSUB1 4}
                   [[1 2]]
                   {FAPPEND [{FHANOI [1 3 2 {FSUB1 4}]}
                            {FCONS [[1 2]
                                    {FHANOI [3 2 1{FSUB1 4}]}
                                     ]
                   }]}
    ))
```

とすれば

(HANOI (4)) ⟹ ((a c) (a b) (c d) (a c) ... (c b))

が得られる.